Lecture 1:

Overview of Fortran 90

Fortran Evolution

History:

- FORmula TRANslation.
- first compiler: 1957.
- first official standard 1972: ‘Fortran 66’.
- updated in 1980 to Fortran 77.
- updated further in 1991 to Fortran 90.
- next upgrade due in 1996 - remove obsolescent features, correct mistakes and add limited basket of new facilities such as `ELEMENTAL` and `PURE` user-defined procedures and the `FORALL` statement.
- Fortran is now an ISO/IEC and ANSI standard.

Design Goals

A compromise between:

- Fortran 77 as a subset;
- efficiency;
- portability;
- regularity;
- ease of use;

Drawbacks of Fortran 77

Fortran 77 was limited in the following areas,

1. awkward 'punched card' or 'fixed form' source format;
2. inability to represent intrinsically parallel operations;
3. lack of dynamic storage;
4. non-portability;
5. no user-defined data types;
6. lack of explicit recursion;
7. reliance on unsafe storage and sequence association features.

Fortran 90 New features

Fortran 90 supports,

1. free source form;
2. array syntax and many more (array) intrinsics;
3. dynamic storage and pointers;
4. portable data types (`KIND`s);
5. derived data types and operators;
6. recursion;
7. `MODULE`s

- procedure interfaces;
- enhanced control structures;
- user defined generic procedures;
- enhanced I/O.

Source Form

Free source form:

- ☐ 132 characters per line;
- ☐ extended character set;
- ☐ '`!`' comment initiator;
- ☐ '`&`' line continuation character;
- ☐ '`;`' statement separator;
- ☐ significant blanks.

New Style Declarations and Attributing

Can state `IMPLICIT NONE` meaning that variables must be declared.

Syntax

$< \textit{type} >$ [,$< \textit{attribute}\text{-list} >$] [::]&
    $< \textit{variable}\text{-list} >$ [=$< \textit{value} >$]

The are no new data types. (If $< \textit{attribute}\text{-list} >$ or =$< \textit{value} >$ are present then so must be ::.)

The following are all valid declarations,

```
 SUBROUTINE Sub(x,i,j)
  IMPLICIT NONE
  REAL, INTENT(IN) :: x
  LOGICAL, POINTER :: ptr
  REAL, DIMENSION(10,10) :: y, z(10)
  CHARACTER(LEN=*), PARAMETER :: 'Maud''dib'
  INTEGER, TARGET :: k = 4
```

The `DIMENSION` attribute declares a 10×10 array, this can be overridden as with `z`.

New Control Constructs

- `IF` construct names for clarity (new relational and logical operators too),

```
zob: IF (A > 0) THEN
       ...
     ELSEIF (A == -1) THEN zob
       ...
     ELSE zob
 chum: IF (c == 0 .EQV. B >= 0) THEN
         ...
       ENDIF chum
       ...
     ENDIF zob
```

- `SELECT CASE` for integer and character expressions,

```
SELECT CASE (case_expr)
 CASE(1,3,5)
  ...
 CASE(2,4,6)
  ...
 CASE(7:10)
  ...
 CASE(11:)
  ...
 CASE DEFAULT
  ...
END SELECT
```

New Control Constructs

- `DO` names, `END DO` terminators, `EXIT` and `CYCLE`,

```
outa: DO i = 1,n
 inna: DO j = 1,m
        ...
       IF (X == 0) EXIT
        ...
       IF (X < 0) EXIT outa
        ...
       IF (X > 10) CYCLE inna
        ...
       IF (X > 100) CYCLE outa
        ...
      END DO inna
     END DO outa
```

- `DO WHILE` but this superseded by `EXIT` clause.

New Procedure Features

- internal procedures,

```
SUBROUTINE Subby(a,b,c)
 IMPLICIT NONE
  ...
  CALL Inty(a,c)
   ...
CONTAINS
 SUBROUTINE Inty(x,y)
  ...
 END SUBROUTINE Inty
END SUBROUTINE Subby
```

- `INTENT` attribute specify how variables are to be used,

```
INTEGER FUNCTION Schmunction(a,b,rc)
 IMPLICIT NONE ! New too
 REAL, INTENT(IN) :: a
 REAL, INTENT(INOUT) :: b
 INTEGER, INTENT(OUT) :: rc
  ...
END FUNCTION Schmunction ! New END
```

New Procedure Features

- `OPTIONAL` and keyword arguments,

```
SUBROUTINE Schmubroutine(scale,x,y)
 IMPLICIT NONE ! Use it
 REAL, INTENT(IN) :: x,y ! New format
 REAL, INTENT(IN), OPTIONAL :: scale
 REAL :: actual_scale
  actual_scale = 1.0
  IF (PRESENT(scale)) actual_scale = scale
  CALL Plot_line(x,y,actual_scale)
END SUBROUTINE Schmubroutine ! Neater
```

called as

```
CALL Schmubroutine(x=1.0,y=2.0)
CALL Schmubroutine(10.0,1.0,2.0)
```

- Explicit recursion is permitted,

```
RECURSIVE SUBROUTINE Factorial(N, Result)
 IMPLICIT NONE
 INTEGER, INTENT(IN)    :: N
 INTEGER, INTENT(INOUT) :: Result
  IF (N > 0) THEN
   CALL Factorial(N-1,Result)
   Result = Result * N
  ELSE
   Result = 1
  END IF
END SUBROUTINE Factorial
```

EXTERNAL Procedure Interfaces

- INTERFACE blocks,

```
INTERFACE
 SUBROUTINE Schmubroutine(scale,x,y)
  REAL, INTENT(IN) :: x, y
  REAL, INTENT(IN), OPTIONAL :: scale
 END SUBROUTINE Schmubroutine
END INTERFACE
```

these are mandatory for EXTERNAL procedures with,

- optional and keyword arguments;
- pointer and target arguments;
- new style array arguments;
- array or pointer valued procedures.

New Array Facilities

- arrays as objects,

```
REAL, DIMENSION(10,10) :: A, B
REAL, ALLOCATABLE(:,:) :: C
REAL :: x = 1.0 ! new
 A = 10.0 ! scalar conformance
 B = A    ! shape conformance
```

- elemental operations,

```
B = x*A + B*B
```

- sectioning,

```
PRINT*, A(2:4,2:6:2)
B(:,10:1:-1) = A(:,:)
```

- array valued intrinsics,

```
B = SIN(A)
B(:,4) = ABS(A(:,5))
```

- masked assignment,

```
WHERE (A > 0.0) B = B/A
```

Program Packaging — Modules

- the `MODULE` program unit may contain
  - definitions of user types,
  - declarations of constants,
  - declaration of variables (possibly with initialisation),
  - accessibility statements,
  - definition of procedures,
  - definition of interfaces for external procedures,
  - declarations of generic procedure names and operator symbols,

  the above provides basis of object oriented technology.

- the `USE` statement,
  - names the particular `MODULE`,
  - imports the public objects,

- provides global storage without `COMMON`,

Stack Example

```
MODULE stack
 IMPLICIT NONE
 PRIVATE
 INTEGER, PARAMETER :: stack_size = 100
 INTEGER, SAVE :: store(stack_size), pos = 0
 PUBLIC push, pop
CONTAINS
 SUBROUTINE push(i)
  INTEGER, INTENT(IN) :: i
   IF (pos < stack_size) THEN
    pos = pos + 1; store(pos) = i
   ELSE
    STOP 'Stack Full error'
   END IF
 END SUBROUTINE push
 SUBROUTINE pop(i)
  INTEGER, INTENT(OUT) :: i
   IF (pos > 0) THEN
    i = store(pos); pos = pos - 1
   ELSE
    STOP 'Stack Empty error'
   END IF
 END SUBROUTINE pop
END MODULE stack
```

Rational Arithmetic Example

```
MODULE RATIONAL_ARITHMETIC
 TYPE RATNUM
  INTEGER :: num, den
 END TYPE RATNUM
 INTERFACE OPERATOR(*)
  MODULE PROCEDURE rat_rat, int_rat, rat_int
 END INTERFACE
 PRIVATE :: rat_rat, int_rat, rat_int
 CONTAINS
  TYPE(RATNUM) FUNCTION rat_rat(l,r)
   TYPE(RATNUM), INTENT(IN) :: l,r
   rat_rat%num = l%num * r%num
   rat_rat%den = l%den * r%den
  END FUNCTION rat_rat
  TYPE(RATNUM) FUNCTION int_rat(l,r)
   INTEGER, INTENT(IN) :: l
   TYPE(RATNUM), INTENT(IN) :: r
    ...
   END FUNCTION int_rat
   FUNCTION rat_int(l,r)
    ...
   END FUNCTION rat_int
 END MODULE RATIONAL_ARITHMETIC
 PROGRAM Main;
  USE RATIONAL_ARITHMETIC
  INTEGER :: i = 32
  TYPE(RATNUM) :: a,b,c
   a = RATNUM(1,16); b = 2*a; c = 3*b
   b = a*i*b*c; PRINT*, b
 END PROGRAM Main
```

User Defined Entities

- Define Type

```
TYPE person
 CHARACTER(LEN=20) :: name
 INTEGER :: age
 REAL  ::  height
END TYPE person
TYPE couple
 TYPE(person) :: he, she
END TYPE couple
```

- Declare structure

```
TYPE(person) :: him, her
TYPE(couple) :: joneses
```

- Component selection

```
him%age, her%name, joneses%he%height
```

- Structure constructor

```
him = person('Jones', 45, 5.8)
them = couple(person(...),person(...))
```

Operators and Generics

- Overloaded operators and assignment

```
INTERFACE OPERATOR (+)
 ... ! what + means in this context
END INTERFACE ! OPERATOR (+)
INTERFACE ASSIGNMENT (=)
 ... ! what = means in this context
END INTERFACE ! ASSIGNMENT (=)
 ...
joneses = him+her
```

- Defined operators

```
INTERFACE OPERATOR (.YOUNGER.)
 ... ! what .YOUNGER. means
END INTERFACE ! OPERATOR (.YOUNGER.)
 ...
IF (him.YOUNGER.her) ...
```

- Generic interfaces (intrinsic and user defined),

```
INTERFACE LLT
 ... ! what LLT means in this context
END INTERFACE ! LLT
INTERFACE My_Generic
 ... ! what My_Generic means in this context
END INTERFACE ! My_Generic
 ...
IF (LLT(him,her)) ...
```

Pointers

- Objects declared with the `POINTER` attribute

  ```
  REAL, DIMENSION(:,:), POINTER :: pra, prb
  ```

  `pra` is a descriptor for a 2D array of reals,

- objects to be referenced must have `TARGET` attribute,

  ```
  REAL, DIMENSION(-10:10,-10:10), TARGET :: a
  ```

- a pointer is associated with memory by allocation,

  ```
  ALLOCATE(prb(0:n,0:2*n*n),STAT=ierr)
  ```

- pointer assignment,

  ```
  pra => a(-k:k,-j:j)
  ```

  `{\tt pra} is now an alias for part of {\tt a}.`

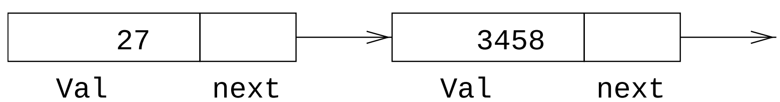
- pointers are automatically dereferenced, in expressions they reference the value(s) stored in the current target,

  ```
  pra(15:25,5:15) = pra(10:20,0:10) + 1.0
  ```

Pointers and Recursive Data Structures

- Derived types which include pointer components provide support for recursive data structures such as linked lists.

```
TYPE CELL
 INTEGER :: val
 TYPE (CELL), POINTER :: next
END TYPE CELL
```



- Assignment between structures containing pointer components is subtlely different from normal,

```
TYPE(CELL) :: A
TYPE(CELL), TARGET :: B
A = B
```

is equivalent to:

```
A%val = B%val
A%next => B%next
```

Parameterised Data Types

- Intrinsic types can be parameterised to select accuracy and range of the representation,

- for example,

```
INTEGER(KIND=2) :: i
INTEGER(KIND=k) :: j
REAL(KIND=l) :: x
```

  where `k` and `m` are default integer constant expressions and are called kind values,

- can have constants

```
24_2, 207_k, 1.08_l
```

- `SELECTED_INT_KIND`, `SELECTED_REAL_KIND` can be parameterised and return kind value of appropriate representation. This gives portable data types.

```
INTEGER, PARAMETER :: k = SELECTED_INT_KIND(2)
INTEGER, PARAMETER :: l = SELECTED_REAL_KIND(10,68)
```

- a generic intrinsic function `KIND(object)` returns the kind value of the object representation:

  - `KIND(0.0)` is kind value of default `REAL`.

  - `KIND(0_k)` is `k`.

New I/O Features

- normal Fortran I/O always advances to the next record for any `READ` or `WRITE` statement,

- Fortran 90 supports non-advancing form of I/O added,

  ```
  WRITE(...,ADVANCE='NO',...) a
  ```

  appends output characters to the current record and

  ```
  READ(...,ADVANCE='NO',...) a
  ```

  reads from the next available character in a file

  ```
  READ(...,ADVANCE='NO',EOR=99,SIZE=nch) a
  ```

  detects end of record and `nch` will contain the number of characters actually read.

Advantages of Additions

Fortran 90 is:

- more natural;
- greater flexibility;
- enhanced safety;
- parallel execution;
- separate compilation;
- greater portability;

but is

- larger;
- more complex;

Language Obsolescence

Fortran 90 has a number of features marked as obsolescent, this means,

- they are already redundant in Fortran 77;
- better methods of programming already existed in the Fortran 77 standard;
- programmers should stop using them;
- the standards committee's intention is that many of these features will be removed from the next revision of the language, Fortran 95;

Obsolescent Features

The following features are labelled as obsolescent and will be removed from the next revision of Fortran, Fortran 95,

- the arithmetic `IF` statement;
- `ASSIGN` statement;
- `ASSIGN`ed `GOTO` statements;
- `ASSIGN`ed `FORMAT` statements;
- Hollerith format strings;
- the `PAUSE` statement;
- `REAL` and `DOUBLE PRECISION` `DO`-loop control expressions and index variables;
- shared `DO`-loop termination;
- alternate `RETURN`;
- branching to an `ENDIF` from outside the `IF` block;

Undesirable Features

- fixed source form layout - use free form;
- implicit declaration of variables - use `IMPLICIT NONE`;
- `COMMON` blocks - use `MODULE`;
- assumed size arrays - use assumed shape;
- `EQUIVALENCE` statements;
- `ENTRY` statements;
- the computed `GOTO` statement - use `IF` statement;